Algoritma Greedy dan Depth First Search dalam Kode Huffman Untuk Kompresi Ukuran File

Fikri Khoiron Fadhila - 13520056
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13520056@std.stei.itb.ac.id

***Abstrak*— Makalah ini membahas mengenai enkoding dan decoding dalam algoritma pengkodean Huffman. Urutan yang akan dibahas adalah pengkodean Huffman, algoritma greedy, Huffman encoding, algoritma depth first search, dan Huffman decoding. Semua penjelasan menunjukkan Huffman encoding menggunakan algoritma greedy dan Huffman decoding menggunakan algoritma depth first search.**

Kata kunci—DFS, greedy, huffman, algoritma, kompresi

I. KODE HUFFMAN

Kode Huffman merupakan kode yang sangat popular untuk mengkodekan dan merepresentasikan data dengan memori yang kecil untuk menyimpan data. Pengkodean Huffman dibuat untuk mengurangi redudansi / duplikat data pada sebuah file. Kode hufman telah dikembangkan oleh David A. Huffman Ketika dia menjadi seorang mahasiswa Ph.D di MIT, dan telah dipublikasikan pada tahun 1952 pada sebuah paper berjudul "A Method for the Construction of Minimum-Redundancy Codes".

A. Enkoding Karakter

Satu file disusun oleh banyak bits untuk merepresentasikan isinya. Bits tersebut tekelompokkan menjadi kumpulan himpunan dari bits. Bits yang terkelompok ini merepresentasikan sesuatu yang dapat dimengerti dalam Bahasa manusia. Sebagai contohnya sebuah karakter 'a' dapat direpresentasikan oleh 01000001. Mengganti representasi dari bits menjadi sebuah karekter disebut dengan skema enkoding karakter. Representasi dari karakter yang dapat dibaca manusia sangat bergantung kepada kode yang digunakan. Misalnya ASCII dan UNICODE memiliki representasi yang berbeda pada setiap karakternya.

B. Redundansi Data Statistik

Sebuah karakter bisa terdapat dalam sebuah file lebih dari satu kali. Contohnya pada string "algoritma", data dari karakter 'a' terdapat didalam string dua kali dan karena itulah data tersebut disimpan dua kali. Menyimpan sebuah karakter lebih dari satu kali itulah yang kita sebut sebagai redundansi data. Redundansi data ini membuang-buang memori untuk menyimpan data yang sama lebih dari satu kali, meskipun begitu, karakter butuh dituliskan lebih dari sekali untuk bisa menyampaikan informasi yang lengkap. Ini merupakan hal yang harus diatasi oleh pengkodean Huffman untuk menyimpan informasi dengan cara yang efisien dan penggunaan memori yang minimum.

C. Kode Huffman

Ide dari kode Huffman yaitu menggunakan frekunsi dari data didalam sebuah file. Data yang memiliki frekuensi tertinggi akan dikodingkan menjadi bit terpendek dan data dengan frekuensi terendah akan dikodingkan menjadi bit terpanjang. Hal ini berarti penggunaan Panjang variable dari data sebagai lawan dari Panjang data yang fixed (tetap) yang ada di dalam data mentah. Panjang data yang fixed akan memiliki Panjang yang sama meskipun memiliki frekuensi yang berbeda-beda. Ini akan menyebabkan informasi membutuhkan memori yang lebih besar.

Pengkonversian bit ditentukan dengan membuat sebuah struktur pohon data. Setiap simpul dati atas ke bawah akan engdung data dari yang paling sering muncul sampai data yang paling jarang muncul. Konvensi dari penulisan struktur pohon ini, child sebelah kiri akan diisi dengan '0' dan child sebelah kanan akan diisi dengan '1'. Pohon akan terus merentang sampai mencapai daun yang hanya mengandung satu data. Jadi, sebuah simpul dalam pohon tersebut akan selalu memiliki 2 buah children.

Pengkonversian bit juga bergantung kepada jumlah dari data yang unik. Jika terdapat dua data, kita hanya membutuhkan '0' dan '1'. Jika terdapat tiga data, kita membutuhkan '0', '10', dan '11'. Semakin banyak data maka semakin banyak juga bit yang dibutuhkan. Dibandingkan dengan 8 bit extended ASCII binary code yang sering digunakan, kode Huffman menghemat lebih banyak space/memori.

Langkah-langkah dalam pengkodean Huffman, misalkan diberikan sebuah string:

1. Pertama hitung frekuensi dari setiap karakter yang ada dalam string
2. Kemudian gambar pohon Huffman berdasarkan frekuensi karakter dari bawah ke atas menggunakan algoritma greedy.
3. Representasikan setiap karakter di pohon berdasarkan konvensi bit.

4. Konversi string yang diberikan menjadi bits baru yang direpresentasikan menggunakan kode Huffman untuk setiap karakter.

II. Algoritma Greedy

A. Definition

Diberikan sebuah permasalah dan temukan solusinya yang terbaik. Salah satu dari banyak cara yang dapat dilakukan yaitu dengan algoritma greedy yang mana menyelesaikan permasalahan langkah demi langkah. Algoritma greedy adalah sebuah algoritma yang memilih solusi terbaik pada setiah tahapnya, yang biasa disebut sebagai optimum local, dengan harapan mendapatkan solusi terbaik dari permasalahan tersebut yang disebut sebagai optimum global

Pilihan dibuat dalam setiap tahap merupakan solusi dari tahap tersebut tanpa menghitung konsekuensi kedepannya. Ini yang membuat algoritma greedy tidak akan selalu memberikan solusi terbaik dalam sebuah masalah karena solusi terbaik dari tahap sekarang tidak selalu menjadi solusi yang terbaik.

Pilihan yang sudah dibuat tidak dapat dibuah, sebuah elemen yang dipilih sebagai solusi terbaik dalam tahap sekarang dan ditambahkan ke dalam solusi akan tersimpan di dalam solusi sampai akhir. Solusi terbaik dalam setiap taham dapat dibuat dengan semua informasi yang diberikan untuk setiap elemen di himpunan kandidat. Sebagai contoh, dalam sebuah himpunan kandidat dari elemen-elemen memiliki informasi A dan informasi B. Pilihan terbaik dapat dibuat menghitung informasi A greedy oleh A atau menghitung informasi B greedy oleh B.

B. Elemen-elemen Algoritma Greedy

Umumnya greedy memiliki 5 elemen:

1. Himpunan kandidat

   Berisi kandidat yang akan dipilih pada setiap langkahnya, misalnya simpul/sisi di dalam graf, task, koin, benda, karakter.

2. Himpunan solusi

   Berisi kandidat yang sudah dipilih.

3. Fungsi seleksi

   Memilih kandidat berdasarkan strategi greedy tertentu. Strategi greedy ini bersifat heuristic.

4. Fungsi kelayakan

   Memeriksa apakah kandidat yang dipilih dapat dimasukkan ke dalam himpunan solusi (layak atau tidak).

5. Fungsi objektif

   Memaksimumkan atau meminimumkan nilai ke dalam himpunan solusi.

Dengan menggunakan elemen-elemen umum dari algoritma greedy, algoritmanya dapat dibangun sebagai berikut

1. Menginisialisasi himpunan solusi dengan himpunan kosong
2. Memilih sebuah nilai dari himpunan kandidat menggunakan fungsi seleksi.
3. Menghapus nilai yang dipilih dari himpunan kandidat
4. Mengecek apakah nilai yang dipilih dapat ditambahkan ke dalam solusi atau tidak menggunakan fungsi kelayakan.
5. Jika nilai yang dipilih dapat ditambahkan menjadi solusi, tambahkan ke solusi.
6. Ulangi Langkah nomor 2 sampai himpunan kandidat kosong
7. Jika himpunan kandidat kosong, himpunan solusi merupakan solusi dari masalah tersebut.

III. Encoding kode huffman

Membuat pohon Huffman setelah menghitung frekuensi dari semua karakter yang muncul dari string yang diberikan merupakan salah satu contoh dari algori8tma greedy. Pohon Huffman dibentuk dari bawah ke atas dengan menggabungkan 2 data dengan frekuensi terendah menjadi satu lalu dilanjutkan keatas.

Elemen-elemen greedy yang membentuk pohon Huffman diantaranya

1. Himpunan kandidat

   Pohon-pohon yang masing-masing merupakan satu simpul yang mengandung masing-masing karakter yang muncul dari string yang diberikan beserta dengan frekuensinya

2. Himpunan solusi

   Pohon Huffman yang mengandung semua delemen dari himpunan kandidat.

3. Fungsi seleksi

   Sebuah fungsi yang memberikan dua pohon dengan frekuensi karakter simpul akar yang minimum.

4. Fungsi kelayakan

   Semua elemen yang berada di himpunan kandidat pasti layak.

5. Fungsi objektif

   Prosedur yang mengkombinasikan dua pohon yang diberikan oleh fungsi seleksi.

Dengan elemen-elemen diatas. Algoritma pembentukan pohon Huffman dapat dibentuk sebagai berikut:

1. Menginisialisasi himpunan solusi dengan himpunan kandidat.
2. Memilih dua pohon dari himpunan kandidat.
3. Menggabungkan dua pohon menjadi satu dengan frekuensi simpul akar merupakan jumlah dari frekuensi dua simpul akar dari pohon yang digabungkan/

4. Ulangi Langkah 2 dan 3 hingga hanya tersisa satu pohon.

Sebuah contoh dari algoritma diatas menggunakan informasi yang diberikan

Tabel 1. Tabel Huffman 1

Karakter	A	B	C	D	E
Frekuensi	15/39	7/39	6/39	6/39	5/39

Dengan fungsi seleksi yaitu memilih dua pohon yang memiliki frekuensi paling rendah. Dua pohon dengan frekuensi terendah yaitu D dan E. Dua pohon tersebut disatukan menjadi sebuah pohon dengan simpul akar merupakan frekuensi yang merupakan penjumlahan dari simpul akar D dan simpul akar E sehingga simpul akar dari si pohon baru yaitu 11/39. Langkah pertama akan membuat tabel menjadi sebagai berikut

Tabel 2. Tabel Huffman 2

Karakter	A	B	C	DE
Frekuensi	15/39	7/39	6/39	11/39

Dengan fungsi seleksi yaitu memilih dua pohon yang memiliki frekuensi paling rendah. Dua pohon dengan frekuensi terendah yaitu B dan C. Dua pohon tersebut disatukan menjadi sebuah pohon dengan simpul akar merupakan frekuensi yang merupakan penjumlahan dari simpul akar B dan simpul akar C sehingga simpul akar dari si pohon baru yaitu 13/39. Langkah kedua akan membuat tabel menjadi sebagai berikut

Tabel 3. Tabel Huffman 3

Karakter	A	BC	DE
Frekuensi	15/39	13/39	11/39

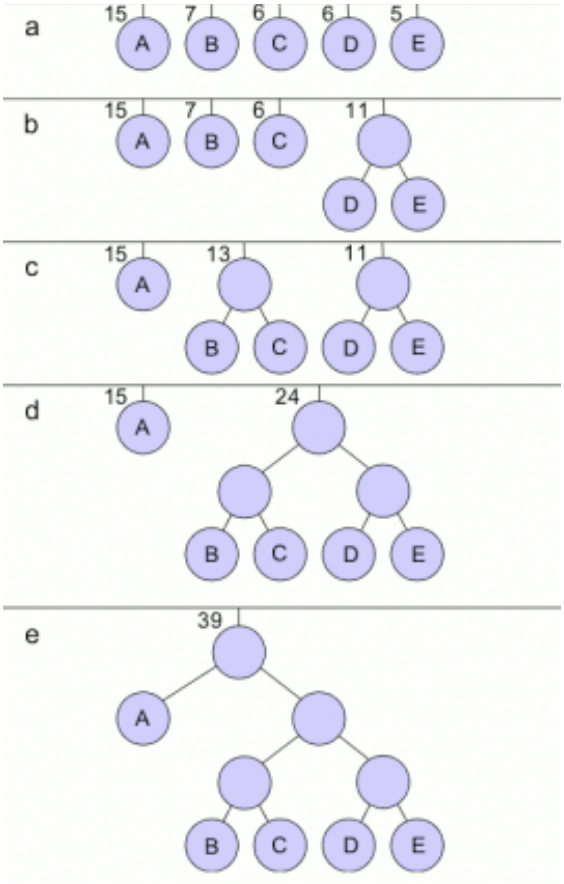
Dengan fungsi seleksi yaitu memilih dua pohon yang memiliki frekuensi paling rendah. Dua pohon dengan frekuensi terendah yaitu BC dan DE. Dua pohon tersebut disatukan menjadi sebuah pohon dengan simpul akar merupakan frekuensi yang merupakan penjumlahan dari simpul akar BC dan simpul akar DE sehingga simpul akar dari si pohon baru yaitu 24/39. Langkah kedua akan membuat tabel menjadi sebagai berikut

Tabel 4. Tabel Huffman 4

Karakter	A	BCDE
Frekuensi	15/39	24/39

Kedua pohon A dan pohon BCDE akan dipilih menjadi pohon baru ABCDE dengan simpul akar berupa frekuensi dengan nilai 39/39. Algoritma akan berhenti karena jumlah dari pohon yang tersisa yaitu satu pohon.

Algoritma diatas akan lebih mudah dimengerti bila direpresentasikan menggunakan pohon.



Gambar 1. Pohon Huffman

Algoritma akan mengonversi karakter menjadi kode Huffman sebagai berikut

Tabel 5. Hasil Pengkodean Huffman

Karakter	Kode ASCII	Kode Huffman
A	0100 0001	0
B	0100 0010	100
C	0100 0011	101
D	0100 0100	110
E	0100 0101	111

Jika kita menggunakan Panjang data yang tetap (fixed), kita membutuhkan 312 bit. Sedangkan jika kita menggunakan kode Huffman data hanya memiliki Panjang 87 bit. Jadi kode Huffman menghemat 225 bit, yang mana ini merupakan memori/ruang yang besar dimana 72.11% dari ruang. Rata-rata code Huffman akan menghemat 20% sampai 30% ruang/memori.

IV. DEPTH FIRST SEARCH

Terdapat dua cara untuk menelusuri graf, yaitu breadth first search dan dept first search.

Breadth first search adalah algoritma pencarian yang dimulai dari simpul akar lalu menelusuri simpul tetangganya. Untuk semua simpul tetangga terdekat, penelurusan simpul tetangganya akan berlanjut sampai selesai.

Dept first search adalah algoritma pencarian yang dimulai dari simpul akar dan meelusuri setiap cabang sejauh yang bisa dicapai. Algoritma ini dapat dimodifikasi menggunakan backtracking yang mana menelusuri node sebelumnya Ketika goal tidak dapat dicapai.

Dept first search dapat diimplementasikan secara rekursif dan tidak rekursif. Berikut ini adalah algoritma DFS seacara rekursif:

1. Mulai pencarian dari simpul akar
2. Cek apakah simpul adalah goal atau bukan
3. Jika simpul adalah goal, maka hentikan pencarian.
4. Jika simpul bukanlah goal, maka pilih satu child.
5. Telusuri pohon child tree secara rekursif.
6. Lakukan Langkah 2 sampai 4 hingga mencapai goal.

Sedangkan algoritma DFS secara tidak rekursif menggunakan queue adalah sebagai berikut:

1. Mulai pencarian dari simpul akar.
2. Tambahkan simpul akar ke antrian.
3. Ambil simpul pertama dari queue.
4. Cek apakah simpul tersebut goal atau bukan, jika iya maka hentikan pencarian.
5. Tambahkan semua simpul tetangganya ke antrian
6. Ulangi Langkah 3 sampai 4 hingga queue kosong.

V. DEKODING KODE HUFFMAN

Proses diatas disebut dengan Enkoding dimana memngonversi informasi direpresentasikan menjadi bentuk yang lain, dalam kode Huffman kasus tersebut mengubah representasi menjadi sebuah struktur data pohon. Proses ini selesai Ketika informasi dapat digunakan untuk berkomunikasi. Untuk mencerna informasi yang kita dapatkan, kita membutuhkan sebuah ‘reverse’ proses yang dinamakan dengan decoding. Dekoding mengonversi informasi Kembali menjadi representasi awal, sehingga receiver bisa mengenali informasi tersebut.

Dekoding sebuah informasi yang dituliskan dalam kode Huffman dapat dicapai dengan du acara. Pertama adalah menelusuri pohon, lalu menuliskan karakter yang ditemukan selama penelusuran, telusuri lagi dari akar pohon. Metode pertama merupakan versi modifikasi dari algoritma dept first search. Kedua yaitu menggunakan sebuah lookup table yang mengandung semua karakter dan bit kodenya. Untuk membandingkan kedua metode diatas, kita akan menggunakan contoh sebelumnya dan mencoba untuk me- decode “01100111” menjadi “ADAE”.

A. Menelusuri Pohon Huffman

Ide dari metode yang pertama yaitu menelusuri pohon Huffman dari bit demi bit dan menuliskan karakter yang ditemukan. Untuk mendekode “01100111” kita menelusuri string dan menemukan ‘0’. Pada DFS kita memulai dari simpul akar. Disini implementasinya akan lebih baik diimplementasikan dengan rekursif DFS. Kita hanya akan melakukan rekursif kepada simpul anak dengan bit yang berkorespondesnsi dengan string. Jadi kita melakukan rekursif kepada akar dari simpul anak yang memiliki nilai ‘0’. Disana kita menemukan karakter A, oleh sebab itu kita menuliskan ‘A’ dan Kembali lagi ke simpul akar.

Penelusuran kedua kita menemukan bit ‘1’ jadi kita melakukan rekursif ke simpul anak yang kanan. Di simpul anak sebelah kanan kita tidak menemukan symbol dikarenakan kita menelusuri ke bit selanjutnya yaitu ‘1’ lalu berpindah ke simpul anak sebelah kanan. Simpul anak sebelah kanan yang kedua juga tidak mengandung karakter jadi kita berpindah ke bit selanjutnya yaitu ‘0’. Berpindah ke simpul anak sebelah kiri kita mendapatkan symbol yaitu D dan Kembali lagi ke akar pohon.

Dengan mengulang proses kita akan mendapatkan karakter ‘A’ dan ‘E’ sebagai karakter selanjutmya. Setelah karakter terakhir, tidak terdapat bit selnjutnya, sehingga bit ‘01100111’ selesai di decoding menjadi “ADAE”

Penelusuran ini lebih baik diimplementasikan secara rekursif dikarenakan kita akan selalu menemukan goal di cabang yang kita telusuri dan tidak diperlukan cabang lain.

Dalam penelusuran ini, kita tidak menyimpan apapun dari string bit, kita hanya menelusurinya berbeda dengan metoda yang kedua.

B. Penelusuran Menggunakan Tabel Pencarian

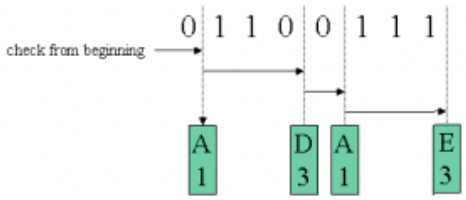
Ide dari metode yang kedua yaitu menggunakan tabel pencarian yang memiliki karakter-karakter dan bit yang sudah diatur. Proses dalam mendekodingnya hanya menelusuri dari string bit, tidak menggunakan pohon Huffman.

Pertama kita membuat sebuah tabel pencarian yang mengandung semua karakter yang masing masing bit nya sudah berkorespondensi dengan karakter. Umumnya tabel berbentuk seperti bael 5. Kemudian kita menelusuri string bit, penelusuran pertama akan bertemu dengan karakter ‘0’ sebagai bit pertama. Kita akan menyimpan bit tersebut lalu melakukan pencarian pada tabel apakah disana terdapat symbol yang direpresentassikan oleh bit ‘0’. Ditemukan ‘A’ terkorespondensi dengan bit ‘0’ di dalam tabel, jadi kita menuliskan karakter ‘A’ lalu dilanjutkan dengan pencarian karakter selanjutnya.

Perbedaan antara metode pertama dan metode yang kedua yaitu, metode pertama hanya menelusuri bit selanjutnya dan tidak Kembali ke akar pohon. Bit selanjutnya yang ditemui yaitu ‘1’, kita menyimpannya lalu mencarinya di tabel. Ternyata tidak adal karakter yang direpresentasikan oleh bit ‘1’ jadi kita menelusuri bit selanjutnya. Kemudian kita menemukan ‘1’, kita menyimpannya yang membuat string bit yang kit acari sekarang yaitu ‘11’. Lakukan pencarian keuda dengan ‘11’

ternyata kita tidak menemukan apa-apa. Telusuri bit selanjutnya dengan string bit sekarang yaitu ‘110’. Dari pencarian akan didapatkan karakter ‘D’. Tulis karakter dan mulai cari karakter selanjutnya. Ulangi proses sampai kita selesai mendekode “01100111” string bit menjadi “ADAE”. Perhatikan bahwa perbedaan kedua antaara metode pertama dan metode kedua yaitu kita menyuimpan bit yang sekarang sedang kita cari.

Modifikasi metode kedua, kita juga bisa menyimpan bit yang direpresentasikan menjadi Panjang dari karakter. Lalu kita dapat menggunakan informasi tersebut untuk memodifikasi penelusuran. Telusuri string bit kita lagi dan akan ditemui bit ‘0’ dan karakter ‘A’. Lalu lanjut ke bit selanjutnya yaitu ‘1’. Perhatikan bahwa semua bit yang direpresentasikan karakter selain karakter ‘A’ memiliki Panjang 3. Jadi, Ketika kita mendapatkan bit ‘1’ setelah mendapatkan sebuah karakter, kita akan menelusuri string tiga kali selama menyimpan setiap bitnya. Setelah itu, kita mencari di tabel untuk mendapatkan karakternya. Kembali ke bit string kita akan mendapatkan ‘110’ dan lakukan pencarian di tabel untuk mendapatkan symbol ‘D’. Ulangi proses tersebut, sampai akhirnya itu mendekode string bit “01100111” menjadi “ADAE”.



Gambar 2 Dekoding Kode Huffman

Kedua metode memiliki cara penggunaan yang berbeda. Menelusuri string bit dan pohon Huffman lebih kompleks daripada hanya menelusuri dari string bit saja. Mencari sebuah tabel dengan banyak data juga kadang banyak masalah. Tergantung dari string bit yang ingin kita dekode. Apakah informasi Panjang atau tidak dan apakah informasi tersebut dibangun dari data yang banyak atau tidak.

VI. Implementasi dan Pengujian Program

Kode program dapat dilihat pada https://drive.google.com/drive/folders/1KSMCXl185jABAGvQRgc4PuAUVN129nmb?usp=sharing
Program akan dimulai dengen mengeksekusi useHuffman.py yang didalamnya sudah memuat path file yang akan di encode dan yang akan di decode.
Dalam mengimplementasikan program kompresi, kunci dari implementasinya yang pertama yaitu membuat struktur data Dictionary yang digunakan untuk menyimpan frekuensi, penggunaan struktur data heap dalam penggabungan pohon dengan menggunakan Class HeapNode yang sudah diimplementasikan, kemudian mengonversi karakter yang dimasukkan menjadi kode biner. Sedikit perbedaan pada implementasi yaitu penambahan padding pada teks yang sudah di encode dengan menjadikan Panjang kodenya kelipatan dari 8.
Di bawah ini merupakan penggalan dari fungsi utama yang akan digunakan untuk melakukan enkoding dan decoding, yang didalamnya akan dilakukan pemanggilan fungsi yang dibutuhkan.

```python
def compress(self):
    filename, file_extension = os.path.splitext(self.path)
    output_path = filename + ".bin"

    with open(self.path, 'r+') as file, open(output_path, 'wb') as output:
        text = file.read()
        text = text.rstrip()

        frequency = self.make_frequency_dict(text)
        self.make_heap(frequency)
        self.merge_nodes()
        self.make_codes()

        encoded_text = self.get_encoded_text(text)
        padded_encoded_text = self.pad_encoded_text(encoded_text)

        b = self.get_byte_array(padded_encoded_text)
        output.write(bytes(b))

    print("Compressed")
    return output_path
```

Gambar 3. Fungsi Encoding

Algoritma encoding secara garis besar akan memanggil fungsi dengan urutan seperti dibawah ini:

1. Membuat struktur data dictionary yang berisi frekuensi
2. Membuat struktur nodes dengan heap memory
3. Menggabungkan nodes
4. Membuat pohon
5. Membuat kode
6. Melakukan encode dari teks
7. Melakukan padding pada bit biner
8. Membuat array byte
9. Mengeluarkan output berupa file biner

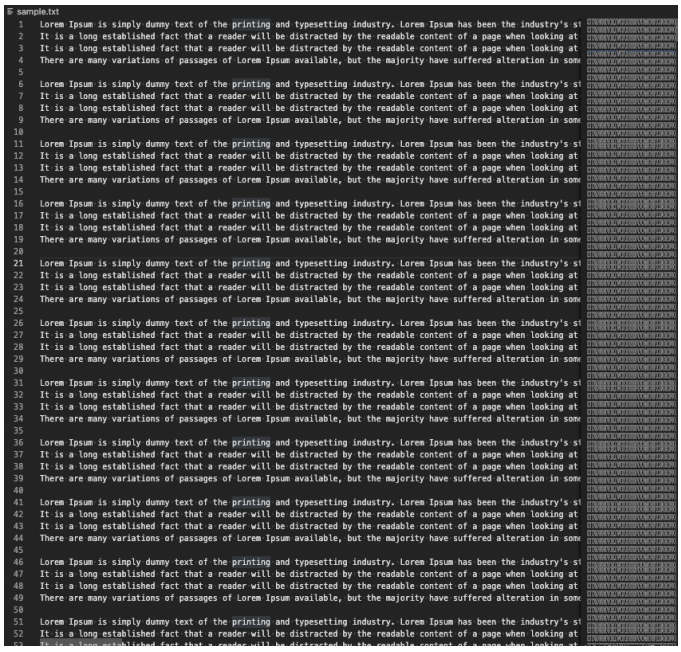
```python
def decompress(self, input_path):
    filename, file_extension = os.path.splitext(self.path)
    output_path = filename + "_decompressed" + ".txt"

    with open(input_path, 'rb') as file, open(output_path, 'w') as output:
        bit_string = ""

        byte = file.read(1)
        while(len(byte) > 0):
            byte = ord(byte)
            bits = bin(byte)[2:].rjust(8, '0')
            bit_string += bits
            byte = file.read(1)

        encoded_text = self.remove_padding(bit_string)

        decompressed_text = self.decode_text(encoded_text)

        output.write(decompressed_text)

    print("Decompressed")
    return output_path
```

Gambar 4. Fungsi Dekoding

Untuk algoritma decoding secara garis besar akan memanggil fungsi dengan urutan seperti dibawah ini:

1. Membaca file biner
2. Menghapus padding
3. Melakukan decode teks
4. Mengeluarkan output berupa file dengan format .txt

Program ini diuji dengan memasukkan file sampel.txt berukuran 715KB.



Gambar 3 File sample.txt

Kemudian kita jalankan program dengan perintah python3 useHuffman.py

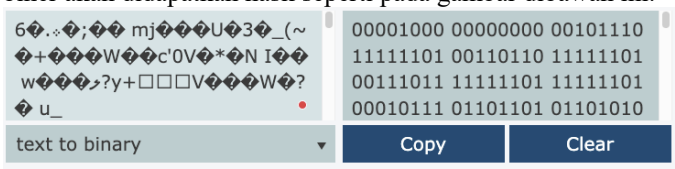
```
(base) fikron@Fikris-MacBook-Air huffman-coding-master 4 % python3 useHuffman.py
Compressed
Compressed file path: sample.bin
Decompressed
Decompressed file path: sample_decompressed.txt
Sukses melakukan dekompresi dengan ukuran file awal:
Ukuran file sebelum encoding: 715332 byte
Ukuran file sesudah encoding: 394017 byte
Persentase perubahan ukuran file: -44.92%
```

Gambar 4 Eksekusi useHuffman.py

File sample.txt akan dienkoding menjadi file Bernama sample.bin yang berisikan kode biner. Apabila file tersebut dibukja akan berisikan seperti pada gambar di bawah.

Gambar 4. Sample.bin

Apabila dilakukan pembacaan file dari teks diatas menjadi kode biner akan didapatkan hasil seperti pada gambar dibawah ini.

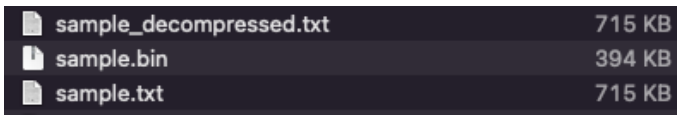


Gambar 5. Translasi teks sample.bin menjadi kode biner

Dari hasil enkoding didapatkan file sample.bin yang berukuran 394KB. Apabila dilakukan perhitungan antara ukuran file sebelum di enkoding dan setelah dienkoding didapatkan presentase perubahan ukuran file sebesear -44.92%.

Akan dilakukan pengujian pada proses decoding file sample.bin. Luaran dari proses decoding berupa file sample_decompress.txt yang berisi seperti pada gambar di bawah ini.

Gambar 6. Hasil decoding sample.bin pada file sample_decompress.txt



Gambar 7. Ukuran file sampel pengujian

File yang didekoding menjadi persis sama dengan file aslinya, tanpa kehilangan data.

VII. Link video dan repository program

Video:
https://youtu.be/5E0x4R4Jz6Q
Repository:
https://drive.google.com/drive/folders/1KSMCXl185jABAGvQRgc4PuAUVN129nmb

VIII. Kesimpulan

Kesimpulan dari makalah ini yaitu

1. Pengkodingan Huffman menggunakan algoritma greedy dengan memilih frekuensi minimum.
2. Dekoding kode Huffman dapat menggunakan DFS yang diimplementasikan secara rekursif ataupun menggunakan tabel pencarian.

3. Pengkodingan Huffman dapat meningkatkan efisiensi penyimpanan data pada file dengan melakukan kompresi ukuran dari file tanpa menghilangkan data asli dari file (lossless).

REFERENCES

[1] http://en.wikipedia.org/wiki/ASCII diakses pada 13 Mei 2022

[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Greedy-(2021)-Bag1.pdf diakses pada 13 Mei 2022

[3] http://en.wikipedia.org/wiki/Character_encoding diakses pada 13 Mei 2022.

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf diakses pada 13 Mei 2022.

[5] https://www.binarytranslator.com/ diakses pada 21 Mei 2022.

[6] https://www.programiz.com/python-programming/methods/built-in/bytearray diakses pada 13 Mei 2022

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Mei 2022

Fikri Khoiron Fadhila - 13520056